ELSEVIER

Theoretical Computer Science 199 (1998) 199–229

Theoretical Computer Science

PHRASE parsers from multi-axiom grammars

Teodor Rus [a], James S. Jones [b,*]

[a] *Department of Computer Science, University of Iowa, Iowa City, IA 52242, USA*
[b] *Division of Science and Mathematics, Graceland College, Lamoni, IA 50140, USA*

Abstract

Multi-axiom grammars (MAG) are alternatives to single-axiom context free grammars (CFG) and all-axiom algebraic grammars (AG) for programming language specification. Neither phrase recognition nor algebraic mechanisms for language processing are supported by CFGs. AGs support algebraic mechanisms for language processing but specify a smaller class of languages. MAGs avoid these limitations. This paper describes a new parsing algorithm developed on this basis which recognizes any phrase in the language. Moreover, it does so by distributing the parsing task among a collection of smaller parsers which handle well-defined layers of the language in a piping manner. These language-layers are determined by the algebraic properties of the MAGs and are described in the paper. Basic definitions are given for multi-axiom grammar and language as well as for algebraic notions of subgrammar, primitive subgrammar, quotient grammar, and grammar/language layer. Algorithms are described to stratify a programming language into a hierarchy of layers, to construct parsers for each layer analogous to LR construction, and to accomplish the overall task of multi-layered parsing in pipeline fashion based on a tokenization which occurs between the language layers. This pipeline parallel process is a model for high speed, left-to-right language translation.

Keywords: Grammar; Subgrammar; Pipeline parsing; Tokenize; Phrase

1. Introduction

Conventional methods for parsing programming languages originated in the early research on context-free grammars [2, 8, 12, 15]. They were developed as operator-precedence methods [9], bounded-context methods [10], LL parsing methods [16], and LR parsing methods [6]. A complete history of the work in this area is found in [1, 18]. The conventional notion of a programming language evolved as the set of valid strings derivable from the single axiom (i.e., the start symbol) of the context free grammar which specified the language. Parsers developed on this basis presume the input to be a monolithic construct, generated from the single axiom. The input of stand-alone

* Corresponding author. E-mail: jsjones@graceland.edu.

0304-3975/98/$19.00
PII S0304-3975(97)00273-9

phrases is, therefore, not supported and the development of incremental programming tools for phrase level activity is inhibited. CYK algorithm ([2], pp. 314–320) performs a dynamic layering of the grammar non-terminals at parse time and thus allows the recognition of phrases embedded within the other phrases [19]. But it requires the specification rules to be in Chomsky normal form and its complexity is $O(n^3)$. In contrast, the parser we are developing here results from the layering of the grammar rules in a preprocessing phase, set no restrictions of the form of specification rules, and its complexity is linear with the length of the input.

In this paper a noncanonical [28] approach to parsing is presented. Its overall goal is to recognize any valid phrase represented by the entire input stream, and to do so in distributed fashion by recognizing the individual subphrase [4] components of that phrase. This represents a vertical fragmentation of the overall parsing process [17]. We will address the two different dimensions of this parsing process:

Parsing by layer is a one-pass (left-to-right) algorithm to recognize phrases, embedded in the input stream, that are associated with a particular language layer.

Multi-pass parsing is an algorithm that manages multiple applications of layered parsing to ultimately recognize the entire input as a meaningful phrase of the language.

The first of these dimensions provides for recognition of embedded phrases whereas the second for recognition of stand-alone phrases, hence the first is a partial solution of the second. Compare this with conventional parsing in which the lexical parser recognizes lexemes embedded in the input stream and the syntax parser recognizes valid, stand-alone programs. Thus, from this point of view the conventional parser operates on only two layers of the language and the overall parsing process is strictly two-pass in nature. Typically, this two-pass nature is obscured by a master–slave relationship between them, i.e., the lexical analyzer being a function called by the syntax analyzer. These two stages have been called phases rather than passes [3]. We use *pass* to stand for a particular parser's complete sweep of its input regardless of the implemented interaction between the layers, whether it be a tightly coupled arrangement (as in master-slave) or a loosely coupled one (as in piping). The approach to parsing presented in this paper deviates from the conventional approach as follows:

- Any valid phrase of the language is recognized in contrast with other (noncanonical) conventional parsers which only recognize valid programs.
- A many layered approach is used in contrast with a conventional two-layered approach.
- The same tool is used to specify each language layer in contrast with using regular expressions for lexical and context-free grammars for syntax layers.
- The same mechanism is used to parse each layer in contrast with using finite-state automata for lexical and push-down automata for syntax parsing.
- The stages of parsing are loosely coupled and pipelined in contrast with conventional designs having tightly coupled lexical and syntax stages.

The parsing algorithm presented in this paper is a simplified noncanonical *SLR* parser [7] which while keeping all the virtues of the noncanonical *NSLR* algorithm

developed by Tai [28] avoids all its difficulties by handling conflict resolution as follows:

1. The language analysis system, *Las* [23], preprocesses the specification grammar and for each production r of the form $A \rightarrow \alpha$ computes the context $\mathscr{C}(r)$, and the noncontext $\mathscr{N}(r)$, which are sets of pairs of strings with the properties:
   (a) if $(x, y) \in \mathscr{C}(r)$ and if $w = \beta_1\ x\ \alpha\ y\ \beta_2$ is a valid construct of the language then the portion α of w is specified by the rule r.
   (b) if $(x, y) \in \mathscr{N}(r)$ and $w = \beta_1\ x\ \alpha\ y\ \beta_2$ is a valid construct of the language then the portion α of w is not specified by the rule r.

   The set $\mathscr{C}(r) \cap \mathscr{N}(r)$ is called the ambiguity set of r and is denoted by $\mathscr{A}(r)$ because, as shown in [14], if the grammar is not ambiguous then $\mathscr{A}(r) = \emptyset$ for each r.
2. The parse table construction, *Algorithm 6*, is a slight modification of the SLR parse table construction developed in [7].
3. The parser, *Algorithm 7*, is an SLR parser that in addition to the usual actions *shift, reduce, error, halt* uses the new actions *pass, accept* and resolves conflicts by *LookAround* method (see Section 4) using the sets $\mathscr{C}(r)$, $\mathscr{N}(r)$, and $\mathscr{A}(r)$ associated with r.

Since *Las* computes the sets $\mathscr{C}(r)$ and $\mathscr{N}(r)$ for each r by first identifying pairs of lengths $0, 1$ and $1, 0$ respectively [22, 23] which are extended to the left and right one grammar symbol at a time, the undecidability of the ambiguity problem translates in nonterminating of the *Las* for some productions. But *Las* can be instructed to terminate after say m extensions to the left and n extensions to the right. This allows the parser to implement a general policy of parsing ambiguous grammars by postponing decisions whenever the context of a rule used in a reduce or an accept action is in $\mathscr{A}(r)$.

We allow a set of non-terminals rather than just one non-terminal to be axioms generating the language specified by a CF grammar. This permits us to develop a meaningful concept of subgrammar that is used to identify particular subgrammars of the original grammar specifying particular sublanguages of the language described by the grammar. In this paper we are concerned only with the hierarchy relationship between the subgrammars of a grammar where a maximal sequence of subgrammars are identified. The key feature here is that a subgrammar of the sequence describes the syntax of the tokens of subgrammar above it by allowing the higher-level subgrammar to reuse the non-terminals of the lower-level subgrammars as its terminals. Thus, we accomplish two goals: (1) use the same mechanism to specify and recognize language constructs of different levels without changing the original grammar, avoiding the difficulties resulting from the extension of BNF rules with *exclusion* and *adjacency-restriction* rules suggested in [26]; (2) construct the parser of the original language from smaller parsers of its sublanguages. However, as one of our reviewers observed, the concept of the subgrammar opens the gate for an algebra of grammars that may lead to new language processing algorithms. The motivation for this approach is two-fold. First, it places language specification on a consistent algebraic framework where complex phrases are constructed from more primitive ones. Second, the independent stages of parsing for

different language layers can be parallelized for high speed translation. In order to do this, we need a concept of a grammar as an algebraic mechanism in which every phrase has meaning, even when it stands alone. Context-free grammars are inappropriate since they promote an all-or-nothing acceptance of entire programs and defining algebraic notions, like subgrammar, are messy. Algebraic grammars [5, 11, 20, 21] do not have these deficiencies but generate a smaller class of languages. This is presented formally in Section 2 along with a formal introduction to a more general specification tool called the multi-axiom grammar [25]. Other fundamental concepts related to the multi-axiom grammars and languages are also presented. Section 3 defines the concept of tokenizing a language and shows how a hierarchy of language layers can be derived. Section 4 shows how to develop a multi-axiom LR parser, called a PHRASE parser, for each layer. Section 5 describes the general algorithm that utilizes the collection of PHRASE parsers in pipelined fashion to accomplish the overall parsing task. Section 6 gives closing comments.

2. Formal notions

This section introduces the formal concepts of multi-axiom grammar and language and discusses their algebraic properties useful for the construction of the parsing algorithms.

2.1. Multi-axiom grammars and languages

A multi-axiom grammar is a generalization of the single-axiom context-free grammar and the all-axiom algebraic grammar. Its formal definition is given below, followed by descriptions of its context-free and algebraic counterparts.

Definition 1. A *multi-axiom grammar* (*MAG*), G, is the quadruple $G=\langle V, \Sigma, P, \mathscr{X}\rangle$ where V is the finite set of *non-terminals*, Σ is the finite set of *terminals*, P is the finite set of *productions*, and $\mathscr{X}$ is the set of *axioms*. The relations among these are: $V \cap \Sigma = \emptyset$, $\mathscr{X} \subseteq V$, and $P \subseteq V \times (V \cup \Sigma)^*$.

Related notations: $\mathscr{V}$ is shorthand for $V \cup \Sigma$ (the *vocabulary*) and $\overline{\mathscr{X}}$ for $V - \mathscr{X}$ (the *non-axioms*). The notation $A \to \alpha$ is used for $r=(A,\alpha)$ and $r \in P$ where A is the *left hand side* of r and is denoted by $lhs(r)$ and α is the *right hand side* of r and is denoted by $rhs(r)$. If $lhs(r) \in \mathscr{X}$ then r is an *axiom rule*, otherwise r is a *non-axiom rule*. *Parse tree*, *derivation*, and the relation between them should be familiar notions to the reader. We use the notation $\Rightarrow$ for derivation step, $\overset{*}{\Longrightarrow}$ for derivation, and $\overset{r}{\Longrightarrow}$ for derivation step by rule $r \in P$. The grammar, G, involved in a derivation may be identified using the notations $\overset{*}{\underset{G}{\Longrightarrow}}$ and $\overset{r}{\underset{G}{\Longrightarrow}}$. A non-terminal, N, *reaches* a symbol X if X occurs in some derivation starting at N; if N reaches the symbol X then X is said to be *reachable* from N.

Definition 2. Given MAG $G=\langle V, \Sigma, P, \mathscr{X}\rangle$ and axiom $A\in\mathscr{X}$, $[A]_G$ (or just $[A]$) is the language generated by A, defined by $[A] = \{\alpha\in\Sigma^* | A\stackrel{*}{\Longrightarrow}\alpha\}$. $[A]$ is also called a *syntax category* of G. $L(G)$ denotes the *multi-axiom language* (*MAL*) generated by G and is defined as the family of sets $L(G)=([A]_A), A\in\mathscr{X}$.

Related definitions: A *context-free grammar* (*CFG*) is a MAG with a single axiom written as $\langle V,\Sigma,P,\{S\}\rangle$ in which $S\in V$. An *algebraic grammar* (*AG*) is a MAG with every non-terminal being an axiom written as $\langle V,\Sigma,P,V\rangle$. A *context-free language* (*CFL*) and an *algebraic language* (*AL*) are the MALs defined by a CFG and AG, respectively.

Lemma 1. *The class of MALs is equivalent to the class of CFLs.*

Proof. Let L be any MAL. L is specified by some MAG, $G=\langle V, \Sigma, P, \mathscr{X}\rangle$. Construct the CFG $G_{CF}=\langle V\cup\{S\},\Sigma,P\cup\{S\rightarrow A \mid A\in\mathscr{X}\},\{S\}\rangle$ where S is a new symbol, i.e., $S\notin\mathscr{V}$. For any $\alpha\in L$ there is a derivation, $A\stackrel{*}{\underset{G}{\Longrightarrow}}\alpha$, for some $A\in\mathscr{X}$. But then $S\Rightarrow A\stackrel{*}{\Longrightarrow}\alpha$ is a derivation in G_{CF} and therefore $\alpha\in L(G_{CF})$. Vice versa, for any $\alpha\in L(G_{CF})$ there is a derivation in G_{CF}, $S\Rightarrow A\stackrel{*}{\Longrightarrow}\alpha$, for some $A\in\mathscr{X}$. But then $A\stackrel{*}{\underset{G}{\Longrightarrow}}\alpha$ and therefore $\alpha\in L$. Thus, $L=L(G_{CF})$ so L is a CFL. Since L was arbitrary, every MAL is a CFL. Clearly, every CFL is an MAL since we define a CFG as a special case of MAG. Hence, the class of MALs and CFLs are equivalent. □

Lemma 2. *The class of ALs is a proper subclass of the class of MALs.*

Proof. By definition an AG is a special case of MAG so every AL is an MAL. To show that the AL class is a proper subclass consider the language ab^*c, an MAL generated by $\langle\{S,B\}, \{a,b,c\}, \{S\rightarrow aBc, B\rightarrow bB, B\rightarrow\varepsilon\}, \{S\}\rangle$. Suppose it is an AL, and thus is specified by some AG, $G=\langle V,\Sigma,P,V\rangle$. Since this language is infinite there is an $N\in V$ such that $N\stackrel{*}{\Longrightarrow}xNy$, for some $x,y\in\Sigma^*$ where either x or y is not the empty string. Since N is an axiom which generates members of ab^*c then either x begins with a or y ends with c. By pumping the same derivations over we get $N\stackrel{*}{\Longrightarrow}xNy\stackrel{*}{\Longrightarrow}xxNyy\stackrel{*}{\Longrightarrow}xxwyy$ for some $xxwyy\in\Sigma^*$. But $xxwyy$ cannot be in ab^*c since it would contain either multiple a's or c's. Hence the assumption that ab^*c is an AL is wrong, so the class of ALs is contained in and smaller than the class of MALs. □

Lemma 3. *The regular sets (REGs) and the ALs are subclasses of the MALs of which there are languages that are both REG and AL, REG but not AL, AL but not REG, and neither REG nor AL.*

Proof. REGs are a subclass of the CFLs [12] and so also a subclass of the MALs by Lemma 1. The ALs are a subclass of the MALs by Lemma 2. Now we show MAL examples for each of the four groups. First, the regular expression b specifies

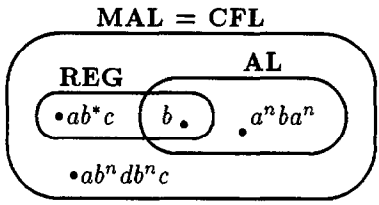

Fig. 1. Classes of languages.

a REG which is also an AL since it is generated by an AG, $\langle\{S\}, \{b\}, \{S \to b\}, \{S\}\rangle$. Second, ab^*c specifies a REG which is not an AL as shown within the proof for Lemma 2. Third, the language $\{a^n ba^n \mid n \geqslant 0\}$ is an AL since it is specified by $\langle\{S\}, \{a,b\}, \{S \to b, S \to aSa\}, \{S\}\rangle$. However, it is not REG as shown in [12]. For that the pumping lemma for regular sets is employed to show that if it were then there would be members of this language with an unbalanced number of a's in it. Finally, the language $\{ab^n db^n c \mid n \geqslant 0\}$ is a MAL generated by $\langle\{S,B\}, \{a,b,c,d\}, \{S \to aBc, B \to bBb, B \to d\}, \{S\}\rangle$. This is not a REG, by the same pumping lemma strategy above to produce a unbalanced number of b's; it is not an AL either, by a similar argument found in the proof of Lemma 2 which shows that ab^*c is not an AL. □

Fig. 1 illustrates the relation between these classes of languages. MAGs are no more powerful than CFGs, with respect to the class of languages they generate. The advantage of a MAG over a CFG is that it handles directly all axiom-derived constructs of the language whereas a CFG merely handles constructs derivable from its only axiom, the start symbol. Also, algebraic notions like subgrammar are difficult to define with CFGs. For instance, it is tempting to refer to the subset of rules which specify the language's lexicon as a subgrammar, but unless we augment it with a single axiom and additional rules then this subgrammar is not a CFG. On the other hand the lexicon is not single-axiom by nature. AGs overcome these problems and, like MAGs, specify a family of syntax categories. However, AGs generate a smaller class of languages than MAGs and cannot generate the complete class of REGs because of the complete absence of non-axioms. Moreover, the language designer loses an important abstraction with AGs, the notion of an intermediate variable for less important constructs (i.e., non-terminals which are not axioms). The MAG, being a generalization of both CFG and AG, gives the language designer flexibility to specify which non-terminals are to be non-axioms.

Definition 3. Let G be a multi-axiom grammar. A *sentence* (aliased *phrase*) is any $\alpha \in \Sigma^*$ such that $A \overset{*}{\Longrightarrow} \alpha$ for some $A \in \mathscr{X}$; a *sentential form* (aliased *phrase form*) is any $\alpha \in \mathscr{V}^*$ such that $A \overset{*}{\Longrightarrow} \alpha$ for some $A \in \mathscr{X}$.

Notational conventions: 1. In this paper, G denotes the MAG $\langle V, \Sigma, P, \mathscr{X}\rangle$. Subscripts and superscripts are consistently applied, for example G_i' denotes $\langle V_i', \Sigma_i', P_i', \mathscr{X}_i'\rangle$, $\overline{\mathscr{X}_i}$ denotes $V_i - \mathscr{X}_i$, and $\mathscr{V}'$ denotes $V' \cup \Sigma'$.

2. $G_{empty}=\langle\emptyset, \emptyset, \emptyset, \emptyset\rangle$ is the *empty grammar* and $L_{empty}=L(G_{empty})=\emptyset$.
3. Unless stated otherwise, we use a and b for terminals, A and B for non-terminals (typically axioms), M and N for non-terminals (typically non-axioms), lower case Greek letters for $\mathscr{V}^*$ strings, X and Y for any symbol in $\mathscr{V}$, and x and y are used in various ways.

An MAG is *reduced* if it is free of *useless symbols*. That is, $\forall X \in \mathscr{V}$, $\exists A \in \mathscr{X}$, $\exists \alpha \in \Sigma^*$, and $\exists v, \omega \in \mathscr{V}^*$, such that $A \overset{*}{\Longrightarrow} vX\omega \overset{*}{\Longrightarrow} \alpha$. By extension, it is also free of useless rules given the specification of P in Definition 1. Any MAG can be transformed into an equivalent MAG that is reduced. The equivalence here is in the sense that the set of derivation sequences from an axiom to a terminal string is the same for both grammars. The transformation of a MAG into an equivalent reduced MAG is a straightforward two-step process similar to conventional ones for CFGs: (1) removal of any non-terminal (and all rules in which it occurs) from which terminal strings cannot be derived; and (2) removal of any symbol (and all rules in which it occurs) which is not reachable from an axiom. Clearly, such a transformation results in an equivalent MAG since equivalence is based only on complete derivations and no elements are eliminated which would affect those in any way. Thus, without loss of generality, throughout the remainder of this paper all MAGs are assumed to be reduced. This simplifies our definitions, proofs, and algorithms.

2.2. Subgrammars and sublanguages

Definition 4. Let G and G' be multi-axiom grammars. G' is a *subgrammar* of G, $G' \preccurlyeq G$, iff $\mathscr{X}' \subseteq \mathscr{X}$, $\overline{\mathscr{X}'} \subseteq \overline{\mathscr{X}}$, $\Sigma' \subseteq \Sigma$, and $P' \subseteq P$. G' is a *proper subgrammar* of G, $G' \prec G$, iff $G' \preccurlyeq G$ and at least one of $\mathscr{X}' \neq \mathscr{X}$, $\overline{\mathscr{X}'} \neq \overline{\mathscr{X}}$, $\Sigma' \neq \Sigma$, or $P' \neq P$ holds.

This subgrammar definition differs from that in [25] with the addition of the $\overline{\mathscr{X}'} \subseteq \overline{\mathscr{X}}$ property. The proper subgrammar definition can be reduced to its necessary conditions, thereby highlighting the intuitive notion that a proper subgrammar of a grammar should have a proper subset of the rules of that grammar. The next lemma does this.

Lemma 4. *$G' \prec G$ if and only if $G' \preccurlyeq G$ and $P' \subset P$.*

Proof. Let G and G' be any multi-axiom grammars in which $G' \prec G$. Clearly, $G' \preccurlyeq G$. Seeking a contradiction, suppose $P' \not\subset P$. Thus $P' = P$ which means that $V' = V$ and $\Sigma' = \Sigma$ (since we assume that G and G' are reduced). By Definition 4 we have $\overline{\mathscr{X}'} \subseteq \overline{\mathscr{X}}$ (i.e., $V' - \mathscr{X}' \subseteq V - \mathscr{X}$) and by substitution $V - \mathscr{X}' \subseteq V - \mathscr{X}$. We also know that $V - \mathscr{X}' \supseteq V - \mathscr{X}$ since $\mathscr{X}' \subseteq \mathscr{X} \subseteq V$. Thus, $V - \mathscr{X}' = V - \mathscr{X}$. Therefore $\mathscr{X}' = \mathscr{X}$, which contradicts the defined properties for $G' \prec G$ and so $P' \not\subset P$ is not true. Hence $P' \subset P$ and the *if part* holds. To prove the converse for the given G and G', suppose that

$G' \preccurlyeq G$ and $P' \subset P$. Obviously, $P' \neq P$ and so, by Definition 4, $G' \prec G$ and the *only if part* holds. □

Lemma 5. *If $G' \preccurlyeq G$, then $L(G') \subseteq L(G)$.*

Proof. Let $G' \preccurlyeq G$. Suppose that $\alpha \in L(G')$. Thus $A \overset{*}{\underset{G'}{\Longrightarrow}} \alpha$ for some $A \in \mathscr{X}'$. Since $P' \subseteq P$ then any derivation in G' is a derivation in G, so $A \overset{*}{\underset{G}{\Longrightarrow}} \alpha$. Furthermore, since $\mathscr{X}' \subseteq \mathscr{X}$ then $A \in \mathscr{X}$. Hence, $\alpha \in L(G)$. □

Lemma 6. *$G' \prec G$ does not imply that $L(G') \subset L(G)$.*

Proof. Let $G = \langle \{A, B\}, \{a\}, \{A \to a, B \to a\}, \{A, B\} \rangle$ and $G' = \langle \{A\}, \{a\}, \{A \to a\}, \{A\} \rangle$. Thus $G' \prec G$, but $L(G') \not\subset L(G)$ since $L(G') = L(G) = \{a\}$. □

Definition 5. Given a MAG G and a phrase $\alpha \in L(G)$, we say that α is an *ambiguous phrase* with respect to G if it has two different parse trees with the same root axiom. G is an *ambiguous grammar* if it specifies ambiguous phrases.

Lemma 7. *If $G' \preccurlyeq G$ and G is not ambiguous then G' is not ambiguous.*

Proof. Let $G' \preccurlyeq G$ in which G is not ambiguous. Assume that G' is ambiguous to seek a contradiction. Then there exists an ambiguous phrase, α, with respect to G' which has two different parse trees with respect to grammar G'. Since all symbols and rules in G' are also in G, then these trees are parse trees for α with respect to grammar G. Thus α is ambiguous with respect to G which means G is ambiguous. This contradicts the initial premise so G' cannot be ambiguous. □

Definition 6. Given a MAG G and a phrase $\alpha \in L(G)$, we say that α is an *overloaded phrase* with respect to G if it has two parse trees with different root axioms. This is equivalent to saying that α is a member of multiple syntax categories in $L(G)$. G is an *overloaded grammar* if it specifies overloaded phrases.

Lemma 8. *If $G' \prec G$ and G is not ambiguous nor overloaded then $L(G') \subset L(G)$.*

Proof. Let $G' \prec G$. Suppose G is neither ambiguous nor overloaded. By Lemmas 4 and 5 $L(G') \subseteq L(G)$. Seeking a contradiction, assume $L(G') = L(G)$. By Lemma 4 there must be some rule $r \in P - P'$ which is not useless since we assume that G and G' are reduced. Hence, there is some $\alpha \in L(G)$ and $A \in \mathscr{X}$ in which $A \overset{*}{\underset{G}{\Longrightarrow}} x \overset{r}{\underset{G}{\Longrightarrow}} y \overset{*}{\underset{G}{\Longrightarrow}} \alpha$ (call this parse tree T_1). Furthermore, since $\alpha \in L(G')$ then there is some $B \in \mathscr{X}'$ in which $B \overset{*}{\underset{G'}{\Longrightarrow}} \alpha$ and since $P' \subset P$ then $B \overset{*}{\underset{G}{\Longrightarrow}} \alpha$ (call this parse tree T_2). Observe that rule r occurs in T_1 but not in T_2 since $r \notin P'$, so T_1 and T_2 are distinct parse trees for α with respect to G. If $A = B$ then α is ambiguous, otherwise α is overloaded. Hence, G is ambiguous or overloaded, which contradicts the assumption. Therefore, $L(G') = L(G)$ is false and so $L(G') \subset L(G)$. □

Corollary 9. *If $G' \prec G$ and $L(G') = L(G)$ then G is either ambiguous or overloaded.*

Proof. The contrapositive of Lemma 8 since $L(G') = L(G)$ iff $L(G') \not\subset L(G)$. □

2.3. Two-layer language specification

To provide a framework for multi-layer heterogeneous language specification, we introduce a two-layer model. In this model, we have a language and a sublanguage of it, in which every phrase of the language is composed of phrases of the sublanguage and primitive symbols. This mimics the "free generation property" in algebra where the language is the collection of well-formed terms and the sublanguage is a collection of free generators. In this section we introduce the concept of a *primitive subgrammar* of a grammar G that generates a sublanguage which has the free generation property. We also define the *most reduced form* which expresses this property.

Definition 7. The set of *primitive symbols* of grammar G is $\overline{\mathscr{X}} \cup \Sigma$.

Definition 8. Given that $G' \preccurlyeq G$ and $N \in V$, we say that N is *completely defined in* G' if every derivation, $N \overset{*}{\underset{G}{\Longrightarrow}} \alpha$, has an identical derivation $N \overset{*}{\underset{G'}{\Longrightarrow}} \alpha$. That is, if $N \overset{*}{\underset{G}{\Longrightarrow}} x \overset{r}{\underset{G}{\Longrightarrow}} y \overset{*}{\underset{G}{\Longrightarrow}} \alpha$ then $r \in P'$.

Definition 9. G' is a *primitive subgrammar* of G, $G' \prec_p G$, if the following properties hold:

1. $G' \preccurlyeq G$;
2. $\forall r \in P$, if $lhs(r) \in \mathscr{X}$ and all symbols in $rhs(r)$ are primitive then $r \in P'$;
3. $\forall N \in \overline{\mathscr{X}'}$, N is completely defined in G';

Definition 10. The language $L(G')$ generated by a primitive subgrammar G' of the multi-axiom grammar G is called a *primitive sublanguage* of the language $L(G)$, $L(G') \prec_p L(G)$.

The primitive subgrammar definition has changed since that of [25] in order to expand the role of non-axioms throughout the layers of the grammar (later in paper), not just the lexical layer. One effect of the current definitions is that G is always a primitive subgrammar of itself, therefore, calling G' a *proper primitive subgrammar* of G will mean $G' \neq G$ and $G' \prec_p G$. Another useful notion is the *smallest primitive subgrammar* of G, which is any primitive subgrammar G' having the smallest $P' \subset P$. Since we assume reduced grammars for these definitions, it can be shown that the only grammar whose smallest primitive subgrammar is G_{empty} is G_{empty} itself.

Example 1. Consider $G = \langle \{E, T, Add, Sgn, Num\}, \{+, -, (,), d\}, P, \{E, T\} \rangle$, a grammar specifying the language of simple expressions. Definition 9 is illustrated by comparing three different subsets of the production rules, P, as shown in Fig. 2.

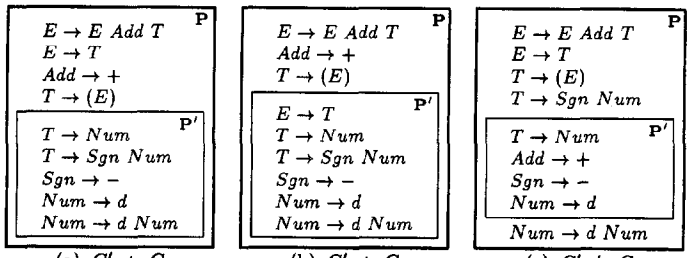

Fig. 2. Three attempts for primitive subgrammar specification.

Fig. 2(a) shows the smallest primitive subgrammar of G since the T–rules must be in G' by property 2 of Definition 9 and the others by property 3. Fig. 2(b) shows another, and larger, primitive subgrammar. In Fig. 2(c), G' is not a primitive subgrammar because it violates all three properties of the definition: $T \rightarrow Sgn\ Num$ is not in G' (violating property 2), *Num* is not completely defined (violating property 3), and G' is not a valid MAG and thus not a subgrammar (violating property 1) because *Add*, *Sgn*, +, and − are unreachable from any axiom in G' (i.e., useless). Before leaving this example, one observes that the rules for a given non-axiom are either all in the primitive subgrammar or none of them are, but the axiom rules may be split between P' and $P-P'$. Also observe that if *Add* were an axiom then $Add \rightarrow +$ would have to be in G', which would establish "+" to be a recognizable phrase in $L(G')$. Hence, the language designer affects what constitutes the smallest primitive subgrammar (and sublanguage) by specifying which non-terminals are axioms and which are non-axioms.

Definition 11. Let $G' \prec_p G$. For any $x \in L(G)$ we call α a *reduced form* of x, in symbols $x \models \alpha$, with respect to G' and syntax category $[A]_G$ if and only if for some $k \geqslant 0$ there exist $s_0, \ldots, s_k \in \Sigma^*$ and $A_1, \ldots, A_k \in \mathscr{X}'$ such that $\alpha = s_0 A_1 s_1 \ldots s_{k-1} A_k s_k$ and $A \overset{*}{\underset{G}{\Longrightarrow}} \alpha \overset{*}{\underset{G'}{\Longrightarrow}} x$.

Definition 12. Let α be a reduced form of x with respect to G' and $[A]_G$. We call α a *most reduced form* (*MRF*) of x, in symbols $x \models_m \alpha$, if and only if $x \models \beta$ implies $\beta \overset{*}{\underset{G'}{\Longrightarrow}} \alpha$.

Lemma 10. *Given that $G' \prec_p G$ and G is not ambiguous, if $x \in L(G)$ then for each $[A]_G$ in which x belongs there exists a unique, α, such that $x \models_m \alpha$.*

Proof. Let $G' \prec_p G$, where G is not ambiguous. Let $x \in L(G)$. Choose any $[A]_G$ in which x belongs. Since G is not ambiguous, x is not ambiguous and so it has a unique parse tree rooted by A. Consider any derivation $A \overset{*}{\underset{G}{\Longrightarrow}} \alpha \overset{*}{\underset{G'}{\Longrightarrow}} x$ ($\alpha = x$ is one such case). This derivation corresponds with a top down traversal of the parse tree and the first

part, $A \overset{*}{\underset{G}{\Longrightarrow}} \alpha$, corresponds with a top down traversal proceeding only as far as the nodes representing the symbols in α. Since the tree has a finite number of traversals, there are a finite number of reduced forms of x, α being one of them (by Definition 11). To show there is a most reduced form of x, it is sufficient to show that for any two reduced forms, a reduced form which is identical or more reduced than each of them exists. Suppose α_1 and α_2 are arbitrary reduced forms. Thus, $A \overset{*}{\underset{G}{\Longrightarrow}} \alpha_1 \overset{*}{\underset{G'}{\Longrightarrow}} x$ and $A \overset{*}{\underset{G}{\Longrightarrow}} \alpha_2 \overset{*}{\underset{G'}{\Longrightarrow}} x$. Consider a top down traversal of this parse tree in which traversing proceeds only as far down as the nodes corresponding with the elements of α_1 or α_2. This traversal corresponds with some $A \overset{*}{\underset{G}{\Longrightarrow}} \alpha_3$. Since the untraversed subtrees below the α_1 and α_2 elements apply rules from G' only, the same is true for α_3. Hence, $\alpha_3 \overset{*}{\underset{G'}{\Longrightarrow}} x$. Since α_3 is a mix of substrings from α_1 and α_2 it has the proper form for Definition 11, and since $A \overset{*}{\underset{G}{\Longrightarrow}} \alpha_3 \overset{*}{\underset{G'}{\Longrightarrow}} x$ it is a reduced form of x. Furthermore, $\alpha_3 \overset{*}{\underset{G'}{\Longrightarrow}} \alpha_1$ and $\alpha_3 \overset{*}{\underset{G'}{\Longrightarrow}} \alpha_2$ since the traversal can continue downward from α_3 to the α_1 or α_2 nodes, respectively, to construct these derivations. So, α_3 is more reduced than α_1 and α_2. Therefore, among the finite set of reduced forms there is a most reduced form, α, such that $x \models_m \alpha$. This α is also unique, since if β is another MRF, then by definition $\alpha \overset{*}{\underset{G'}{\Longrightarrow}} \beta$ and $\beta \overset{*}{\underset{G'}{\Longrightarrow}} \alpha$, and so $\alpha = \beta$ and uniqueness holds. □

Corollary 11. *Given that $G' \prec_p G$ and G is not ambiguous, if $x \in L(G)$ is not an overloaded phrase with respect to G then there is a unique α such that $x \models_m \alpha$.*

Proof. By Lemma 10 there is a unique MRF for each syntax category containing x, but in this case x belongs to only one syntax category so there is only one such MRF. □

Corollary 12. *Given that $G' \prec_p G$ and G is not ambiguous, if G is not overloaded then there is exactly one MRF with respect to G' for any $x \in L(G)$.*

Proof. Extend Corollary 11 to all $x \in L(G)$, none of which are overloaded. □

2.4. Grammar properties for layering

Before we go forward with full grammar layering we must consider the role of non-axioms. We intend for the strings derived from non-axioms to be substring portions of the phrases derived by axioms. This means non-axiom symbols are more basic (or primitive) than axioms. Definitions 7 and 9 follow this intent and specify the notion of primitivity used in this paper. However, this notion of primitivity would not be appropriate for a grammar having non-axioms which can reach axioms, so we restrict the grammars on which layering will be performed to have the following property.

Definition 13. Grammar G is *clean* if it has no non-axioms which reach axioms. This is the same as saying every rule $r \in P$ is a *clean rule*, that is $lhs(r) \in \overline{\mathscr{X}}$ implies there are no axioms in $rhs(r)$.

Any grammar, G_{orig}, can be transformed into a clean grammar, G, which generates the same collection of syntax categories. A sketch of this transformation follows:

1. Set G to G_{orig} initially.
2. If there is some non-axiom rule $N \to \alpha A \beta$ in P where $A \in \mathscr{X}$, then remove A from $\mathscr{X}$ (making it a non-axiom), add a new symbol A' to $\mathscr{X}$ and add $A' \to A$ to P.
3. Repeat step 2 until G is unchanged by it.

The invariant property after each step 2 is that the same syntax categories are generated by G and G_{orig}, albeit the corresponding syntax categories in G might be generated by A' instead of A, but in this case $[A']_G = [A]_{G_{orig}}$. Termination and the clean property are assured since there are fewer unclean rules after each execution of step 2 except for the last one when there are none.

In Section 2.2, the properties for ambiguous and overloaded grammars were defined. The former is a multi-axiom generalization of the conventional CFG notion of ambiguity [1]. The latter is a naturally occurring phenomena in languages (natural or otherwise) in which phrases have different meanings when occurring in different contexts, but in our case this amounts to membership in multiple syntax categories. Overloading is an acceptable form of non-determinism (called ambiguity by linguists [27]) which our algorithms can handle. Ambiguous grammars resulting from ambiguous expression rules, like $E \to E+E \mid E*E \mid a$, could be augmented with explicit priority and associativity rules for the operators (similar to LR solutions) but this paper does not address that in the algorithms. Therefore, we assume in the remainder of this paper that the grammars are clean and unambiguous. Clearly, a subgrammar of a clean grammar is clean since every rule in the grammar is clean. Also, this together with Lemma 7 means that any subgrammar of a clean and unambiguous grammar is clean and unambiguous. The algorithms in the following sections assume clean and unambiguous grammars and maintain these properties in grammar constructions.

3. The grammar hierarchy

Conventional compiler designers use the term *token* for the placeholder representative of lexemes in a programming language. Implicit here is the existence of two grammars which specify adjacent layers of the entire language, i.e., the lexicon and the syntax. This simplifies language design because the syntax grammar specifies a considerably reduced language in which a potentially infinite number of lexemes (such as all identifiers) are factored out of the entire language and replaced by a token (such as *Id*). In Section 3.1 we formalize this for multi-axiom grammars and languages. For that we use a MAG to specify the entire language and allow *any* primitive subgrammar of it to specify the lower layer language which is to be factored out. Thus, we are not restricted to mere lexical categories for the lower layer and so *token* means the placeholder representative for a category of primitive phrases. In symbols, $\sharp A$ denotes the token for syntax category $[A]$. The initial two-layered stratification in Section 3.1 is then refined to a complete multi-layered expansion in Section 3.2.

3.1. Tokenizing a language on two layers

Let G be unambiguous and $G' \prec_p G$. From Lemma 10 in Section 2.3 it is clear that each $x \in L(G)$ can be represented by an expression of the form:

$$s_0\ [A_1]\ s_1 \ldots s_{m-1}\ [A_m]\ s_m$$

where $s_0, \ldots, s_m \in \Sigma^*$, $A_1, \ldots, A_m \in \mathscr{X}'$, and $x \models_m s_0 A_1 s_1 \ldots s_{m-1} A_m s_m$. If G is not overloaded then by Corollary 12 this representation of x is always unique. Otherwise, by Corollary 11 this representation is unique if x is not an overloaded phrase. Either way, the set of language elements that can be obtained from x can be specified by just one string, namely $s_0\ \sharp A_1\ s_1 \ldots s_{m-1}\ \sharp A_m\ s_m$, where the token $\sharp A_i$ represents syntax category $[A_i]$. This replacement is called *tokenizing the string*. Using an appropriate *tokenization process* we can simplify the entire language by tokenizing every sentence of the language. We call this process *tokenizing the language*. The simplification is two-fold: (1) a potential infinite number of primitive phrases are factored out of the language and replaced by their tokens; and (2) a potential infinite number of complex sentences sharing common most reduced forms are replaced by their tokenized strings. To formalize this process, first we define, for a given grammar G, an auxiliary function $reachable_G(X)$ which takes some $X \subseteq V$ and yields the transitive closure of reachable non-terminals and is computed by the following algorithm.

Algorithm 1 *Computes the function* $reachable_G(X)$
1. $\mathscr{N} := X$, where $X \subseteq V$
2. **for** each $N \in \mathscr{N}$ **do**
   for each $N \to \alpha M \beta \in P$ such that $M \in V$ **do**
   $\mathscr{N} := \mathscr{N} \cup \{M\}$
3. **return** $\mathscr{N}$

Tokenizing a language, $L(G)$, with respect to a primitive sublanguage of it is done by first identifying the primitive subgrammar, G', that specifies the primitive sublanguage, factoring it out of the general specification, G, and replacing it in G by token substitutes. Assuming that G and G' are given such that $G' \prec_p G$, the entire process is a grammar operation which computes the *quotient* of G by G', denoted G/G'. This operation is performed by Algorithm 2

Algorithm 2 *Constructs the quotient grammar* $G_q = G/G'$
1. $G_q := G_{empty}$ (i.e., $V_q := \emptyset, \Sigma_q := \emptyset, P_q := \emptyset, \mathscr{X}_q := \emptyset$)
2. **for** each rule r in $P - P'$ **do**
   add r to P_q and if r is an axiom rule then also add $lhs(r)$ to $\mathscr{X}_q$
   update Σ_q, V_q, and $\mathscr{X}_q$, to be consistent with the added rule
3. **for** each axiom rule $A \to \alpha$ in P' **do**
   add $A \to \sharp A$ to P_q and add $\sharp A$ to Σ_q
   add A to both V_q and to $\mathscr{X}_q$

4. $X := V_q - \mathscr{X}_q$ (i.e., X is the set of non-axioms in G_q thus far)
5. **for** each N in reachable$_G(X)$ **do**
   for each rule $N \rightarrow \alpha$ in P' **do**
   add $N \rightarrow \alpha$ to P_q and add N to V_q
   update Σ_q, V_q, and $\mathscr{X}_q$, to be consistent with the added rule

Steps 1–3 of Algorithm 2 set G_q initially to be the same as grammar G excluding the primitive subgrammar rules and substituting token rewriting rules for the primitive syntax categories (axioms in G'). Steps 4 and 5 are needed in cases where a non-axiom occurs in both G_q and G' and therefore must be completely defined in G_q (as it was in G'). Since we assume clean grammars, only non-axiom rules can be added to G_q by step 5. Clearly, G_q specifies the same language as did grammar G, albeit simplified by tokenization. We say that G_q and G' specify adjacent *layers of the language*, with G' specifying the lower layer. Since G_q and G' contain disjoint sets of the axiom rules, we have a partitioning of the original axiom rules. Each partition is associated with a *grammar layer* in correspondence also with a language layer.

3.2. Specifying multiple language layers

The quotient grammar obtained by Algorithm 2 is itself a multi-axiom grammar and if it has a proper primitive subgrammar then we can continue dividing the language into more layers by an iterative process, continuing to partition the axiom rules of the quotient grammar. A language designer specifies the overall grammar, G, and the initial primitive subgrammar, G'. An already familiar specification of the language's lexicon for G' might be used, but any primitive subgrammar of G is valid. Beyond this, the process of layering is an automated one with the goal of maximizing the number of layers. For this, we will need to compute the smallest primitive subgrammar at each iteration, but modified to treat axioms which have been tokenized in previous iterations like primitive symbols in subsequent iterations. Suppose X is the set of non-terminals which are to be treated as primitive symbols (hence $\overline{\mathscr{X}} \subseteq X \subseteq V$). We compute the smallest primitive subgrammar of G with respect to X, denoted $SPS(G,X)$, by Algorithm 3.

Algorithm 3 *Computes the smallest primitive subgrammar of G with respect to X*
1. $G' := G_{empty}$
2. **for** each axiom rule $A \rightarrow \alpha \in P$ such that $\alpha \in (\Sigma \cup X)^*$ **do**
   add $A \rightarrow \alpha$ to P' and add A to both V' and to $\mathscr{X}'$
   update Σ', V', and $\mathscr{X}'$, to be consistent with the added rule
3. $Y := reachable_G(V' - \mathscr{X}')$
4. **for** each $N \in Y$ **do**
   for each rule $N \rightarrow \alpha$ in P **do**
   add $N \rightarrow \alpha$ to P' and add N to V'
   update Σ', V', and $\mathscr{X}'$, to be consistent with the added rule

Step 2 selects the smallest set of axiom rules to satisfy property 2 of Definition 9 provided that members of X are treated as primitives. Since we assume the grammars are clean, step 3 identifies the smallest set of non-axioms which must be in G' and then step 4 selects the smallest set of additional rules to satisfy property 3 of Definition 9.

We are now ready to define what we mean by a layering process that achieves a maximum number of layers. It is based on the idea that the initial G' is set and that the smallest primitive subgrammar at subsequent layers is defined by the *SPS* operation. Given G' and G, where $G' \prec_p G$, the process that performs a complete layering of the language $L(G)$ constructs the longest sequence of pairs of grammars (G'_0, G_0), $(G'_1, G_1), \ldots, (G'_m, G_m)$ such that: $G_0 = G$, $G'_0 = G'$, for all $1 \leqslant i \leqslant m$ it holds that $G_i = G_{i-1}/G'_{i-1}$ and that $G'_i = SPS(G_i, X_i)$ where X_i is $\overline{\mathscr{X}} \cup \mathscr{X}'_0 \cup \cdots \cup \mathscr{X}'_{i-1}$, and $G'_i \prec G_i$, for each $i < m$. This language layering process is carried out by the Algorithm 4.

Algorithm 4 *Construct the grammar hierarchy of G*

1. $G_0 := G$; $G'_0 := G'$; $X := \overline{\mathscr{X}}$; $i := 0$
2. **for** $i = 1, 2, \ldots$ **while** $P'_{i-1} \neq P_{i-1}$ **do**
   - 2.1 $G_i := G_{i-1}/G'_{i-1}$ (by Algorithm 4)
   - 2.2 $X := X \cup \mathscr{X}'_{i-1}$
   - 2.3 $G'_i := SPS(G_i, X)$ (by Algorithm 3)

Step 1 of Algorithm 4 defines the initial layer as given by the user and each step 2 iteration defines a subsequent layer by computing its grammar and primitive subgrammar pair, (G_i, G'_i). Step 2.1 computes G_i, the grammar which specifies the original language tokenized by layers 0 to $i-1$. Step 2.2 adds any axioms which were tokenized in step 2.1 to X, the set of non-terminals to be treated as primitive symbols in subsequent iterations. Step 2.3 computes G'_i to be the smallest primitive subgrammar with respect to X. X is guaranteed to become larger each time, and therefore G'_i is guaranteed to include previously unselected axiom rules. Algorithm 4 terminates when there can be no more layers to compute, that is when grammar G_i and primitive subgrammar G'_i are identical.

Example 2. A multi-axiom grammar and one of its primitive subgrammars

$$G = \langle \{E, T, F, Id, Cn, Sgn, Num, Add, Mul\}, \{+, *, (,), a, d, -\}, P, \mathscr{X} \rangle$$

$$\mathscr{X} = \{E, T, F, Id, Cn\}$$

$$P = \{E \to E\ Add\ T,\ Add \to +,\ E \to T,\ T \to T\ Mul\ F,\ Mul \to *,\ T \to F,$$
$$F \to (E),\ F \to Id,\ F \to Cn,\ Id \to a\ Id,\ Id \to a,\ Cn \to Sgn\ Num,$$
$$Num \to d\ Num,\ Num \to d,\ Sgn \to -,\ Sgn \to \varepsilon\}$$

$$G' = \langle \{Id, Cn, Sgn, Num\}, \{a, -, d\}, P', \mathscr{X}' \rangle$$

$$\mathscr{X}' = \{Id, Cn\}$$

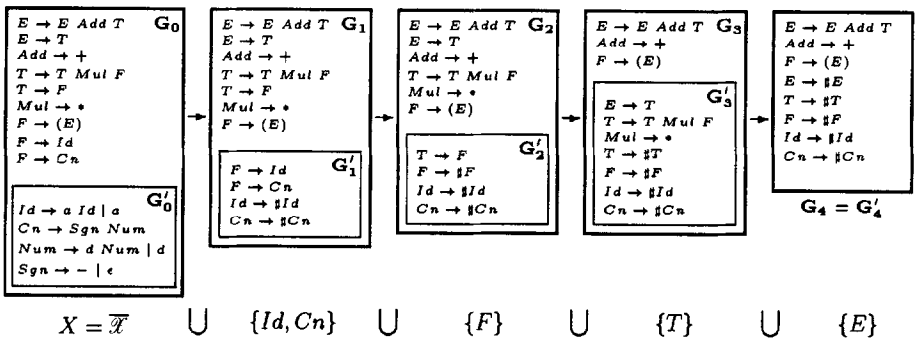

Fig. 3. Grammar layering by Algorithm 4.

$$P' = \{Id \to a\ Id,\ Id \to a,\ Cn \to Sgn\ Num,\ Num \to d\ Num,\ Num \to d,$$
$$Sgn \to -,\ Sgn \to \varepsilon\}$$

Example 2 identifies the grammar and its primitive subgrammar which are used throughout the remainder of the paper. In this example, G specifies a language of arithmetic expressions and G' specifies the data items used to construct such expressions. The result of Algorithm 4 applied on the given G and G' is shown in Fig. 3. One sees that the halt condition is met when $G'_4 = G_4$. Observe in Fig. 3 how the non-axiom rules for *Sgn* and *Num* drop out right away whereas those of *Add* and *Mul* propagate to higher layers. Also observe that not all terminals are handled in the first layer and not all terminals are tokenized (such as the parenthesis). This shows the convenience of this approach to the language designer who may choose to ignore some terminals in the lower language layers, with the effect on parsing being that certain inputs are passed from one stage to the next without processing them in the early stages of parsing. Contrast this with conventional language design in which the specified lexicon covers every terminal and thus the lexical analyzer processes (and consumes) every input character during the lexical phase of parsing.

Dividing the language into layers allows us to divide the task of a parser of the language into many subtasks, reducing its complexity and providing opportunity for parallelism. The remainder of this paper is directed toward a parsing algorithm based on this language layering model.

4. Parsing by layer

Given $G' \prec_p G$, assume that each production r of G is preprocessed by *Las* [23] and is associated with the sets context $\mathscr{C}(r)$, noncontext $\mathscr{N}(r)$, and ambiguity $\mathscr{A}(r)$. The intuitive idea behind PHRASE parsing is: (a) construct the SLR parse table, $PT(G')$, from the production set of G' following an algorithm similar to that presented in [7] that allows $PT(G')$ entries to contain conflicting actions, and (b) develop a noncanonical LR

parsing algorithm [28] controlled by $PT(G')$ that departs from the usual SLR parsing as follows:

1. The parser consumes the entire input from left to right and is always in one of the two overall parsing states: (a) *passive*, where it passes input symbols to the output, and (b) *active*, where it tries to discover a phrase of $L(G')$ embedded in the input.
2. In active state the parser performs on a stack in the LR(0) manner of operation, except that when a conflict in $PT(G')$ involving the rules r_1 and r_2 is encountered the entire context provided by $\mathscr{C}(r_1)$, $\mathscr{N}(r_1)$, $\mathscr{A}(r_1)$ and $\mathscr{C}(r_2)$, $\mathscr{N}(r_2)$, $\mathscr{A}(r_2)$, respectively, is used; in passive state the parser simply copies the contents of the stack or the next input symbol to the output.
3. The parser switches from passive to active state when it receives a symbol from $\mathscr{V}'$; the parser switches from active to passive state when it reaches a decision (i.e., recognizes or fails to recognize a construct from $L(G')$) or when it receives a symbol from $\mathscr{V}\setminus\mathscr{V}'$.

Hence, a PHRASE parser operates on configurations of tuples $\langle Output, Stack, Input\rangle$ and performs reduction steps $\langle Output, Stack, Input\rangle \overset{step}{\Longrightarrow} \langle Output', Stack', Input'\rangle$ maintaining the following *phrase parsing invariant*:

> Let x, α, y be the contents of *Output*, *Stack*, and *Input*, before a reduction step and x', α', y' after the reduction step. Then if $x\ \alpha\ y$ is a phrase form of $L(G)$ then $x'\ \alpha'\ y'$ is a phrase form of $L(G)$.

We use the name PHRASE for this parser because it suggests exactly what the parser is doing, i.e., it recognizes language phrases embedded within other language phrases and at the same time it is an acronym for the parsing actions involved: *Pass*, *Halt*, *Reduce*, *Accept*, *Shift*, *Error*. An earlier version of this parsing algorithm was called a SHARE parser [25] because it performed the actions *Shift*, *Halt*, *Accept*, *Reduce*, *Error* [25]. We further discuss the PHRASE parser in the context of the grammar hierarchy described in Section 3.

4.1. The PHRASE parser

Let (G'_0, G_0), $(G'_1, G_1), \ldots, (G'_m, G_m)$ be the hierarchy constructed by Algorithm 4 from a given G and G' where $G' \prec_p G$, $G = G_0$, and $G' = G'_0$. In this section we present the algorithm that constructs a PHRASE parser of language layer k (for $0 \leqslant k \leqslant m$), that is $L(G'_k)$, a primitive sublanguage of the language $L(G)$. In Section 5 we show how a collection of PHRASE parsers for all layers is used to achieve the overall goal of parsing a sentence from the entire language, $L(G)$, in pipeline fashion. The general idea for PHRASE parsing is illustrated in Fig. 4. A PHRASE parser is a table-driven, bottom-up parser which scans the input string and tokenizes it based on the primitive subgrammar, G'_k, using a stack in a manner similar to every LR parser. PHRASE parsing has similarities with LR parsing [1], though the individual parse tables for each layer will be smaller than a monolithic LR table. The table entries are allowed to contain conflicting actions. Parsing conflicts are however resolved at parsing time by a

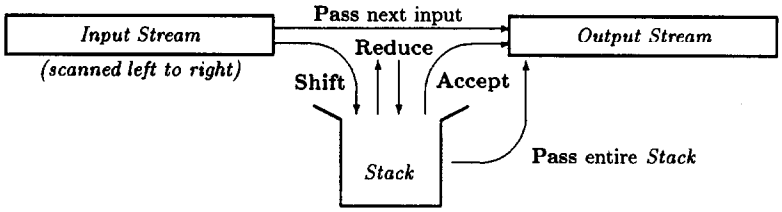

Fig. 4. PHRASE parsing.

special function called *Disambiguate*( ) using the information attached by *Las* to the specification rules. The contents of the stack represents a viable prefix of a potential phrase in $L(G'_k)$. When the parser makes a decision with respect to the validity of the language construct accumulated in its stack the contents of the stack is copied to the output and the parser enters a passive state where it simply copies symbols from the input to the output; the parser enters again an active parsing state when it discovers a symbol in the input that can potentially begin a phrase of the language $L(G'_k)$. With this in view, the different actions taken by the PHRASE parser are:

1. *Pass*, that copies the next input symbol or the contents of the stack to the output.
2. *Halt*, that terminates the parsing process.
3. $Reduce_r$, that replaces the $rhs(r)$, for some $r \in P'_k$, on top of the stack by $lhs(r)$, and continues in the same way as an LR parser does.
4. $Accept_r$, that replaces the $rhs(r)$, for some $r \in P'_k$, on top of the stack by $lhs(r)$, where $lhs(r)$ is an axiom, and enters the passive state.
5. $Shift_k$, that pushes the input onto the stack and thereby advances the input and extends the viable prefix represented by the stack.
6. *Error*, which is handled like a pass action thus postponing the parsing decision.

These actions are performed by a finite state machine (FSM) that has a component similar to the conventional LR(0) parser for recognizing viable prefixes, but multi-axiom in nature. We use the conventional notation to represent *items* (i.e., a "•" denoting a scan position within $rhs(r)$) and the role played by items, item sets, *closure*, and *goto* are all the same as conventionally [1]. The *multi-axiom LR*(0) *item set construction* that generates the item sets $\mathscr{S} = \{S_0, S_1, \ldots, S_n\}$ from the given primitive subgrammar G'_k is specified by Algorithm 5 as follows:

Algorithm 5. Multi-axiom LR(0) item set construction for G'_k

1. $S_0 := closure(\{\ [A \rightarrow \bullet\, \alpha] \mid A \rightarrow \alpha$ is an axiom rule in $G'_k\})$
2. $n := 0$; $\mathscr{S} := \{S_0\}$;
3. **for** *each* $S_i \in \mathscr{S}$ **do**
   for each $X \in \mathscr{V}'_k$ **do**
   $S := closure(\{[A \rightarrow \alpha\ X \bullet \beta] \mid [A \rightarrow \alpha \bullet X\beta] \in S_i\})$
   if $S \notin \mathscr{S}$ **then**
   $n := n + 1$; $S_n := S$; $\mathscr{S} := \mathscr{S} \cup \{S_n\}$;

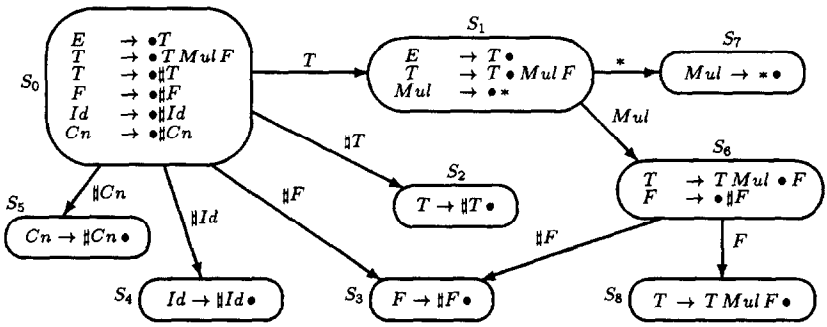

Fig. 5. Multi-axiom LR(0) item sets for G_3'.

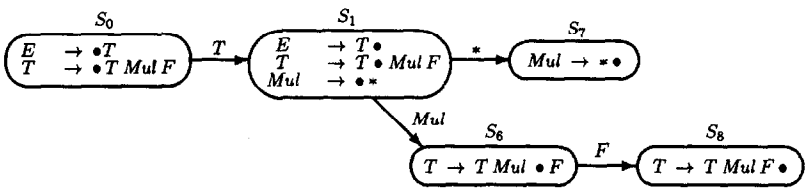

Fig. 6. Simplified FSM for G_3' with reuse of axiom symbols.

The difference between this and conventional methods which captures the multi-axiom nature of the method occurs in step 1 where an unscanned item associated with each axiom rule in G_k' is inserted into the initial item set. The resulting FSM defined by this construction recognizes a viable prefix for any phrase specified by G_k' and so here the prefixes are not prefixes of rightmost sentential forms of the entire input stream, but rather are prefixes of rightmost phrase forms or phrases embedded in the input (which phrase could be the entire input). The item set construction for G_3' (see Fig. 3) is illustrated in Fig. 5.

Up to this point we have been careful to distinguish between an axiom and its respective token with different symbols, i.e., A or $\sharp A$. This was done to keep the formal notions clean, but the distinction is of no consequence to a parser. Nor do we need the tokenizing unit productions, $A \to \sharp A$, which merely act to translate tokens into axioms. Therefore, we reuse the axiom symbols to stand also for their respective tokens. Implicitly, such a symbol represents a token when it occurs in the input or output stream, and represents an axiom otherwise. The result is a much reduced FSM. Fig. 6 shows the reduced FSM in Fig. 5. Notice that symbols *Cn* and *Id* do not occur in the remaining item sets in Fig. 6 because they are irrelevant to parsing at this layer. Therefore, for convenience and with no harm to our methodology we will suppose that no $r \in P_k'$ of the form $A \to \sharp A$ is used in Algorithm 5 and that $\mathscr{V}_k'$ includes only symbols occuring in P_k'. All other symbols from the entire vocabulary, $\mathscr{V} = \mathscr{V}_0$, including begin or end file markers (denoted \$), will be collectively called *Other*.

Constructing the PHRASE parse table proceeds from the collection of item sets constructed by Algorithm 5. When an item set (called an LR parsing state) contains items which call for conflicting actions then we will have multiple actions occurring in the corresponding table entry. Conflicts will be resolved at parsing time by a *disambiguate* action discussed in detail later. Given the layer k grammar, G'_k, for the sublanguage $L(G'_k)$ of $L(G)$, the construction of the PHRASE parse table, $\mathscr{T}_k[0..n-1, 0..m]$, where $m-1$ is the size of the reduced V'_k and n is the number of LR parse states of the FSM constructed by Algorithm 5, is defined by Algorithm 6. The entry $\mathscr{T}_k[i,j]$ records the actions discovered by the Algorithm 6 in the item set i, $0 \leqslant i \leqslant n-1$, when next input symbol is $x_j \in V'_k$, $0 \leqslant j \leqslant m-1$. The column $\mathscr{T}_k[i,m]$, $0 \leqslant i \leqslant n-1$, records the actions discovered by the Algorithm 6 when the next symbol is *Other*, representing any symbol $x \notin V'_k$.

Algorithm 6. *PHRASE parse table construction for* G'_k

1. Construct item sets $\mathscr{S} = \{S_0, S_1, \ldots, S_n\}$ by Algorithm 5 for G'_k
2. $\mathscr{T}_k[i,j] := \emptyset$ for all $S_i \in \mathscr{S}$ and $x_j \in V'_k \cup \{Other\}$
3. **for** each item set $S_i \in \{S_0, \ldots, S_n\}$ **do**
   for each $item \in S_i$ **do**
   3.1. **if** $item = [A \to \alpha \bullet x_j \beta\ ]$ **then**
      3.1.1. $\mathscr{T}_k[i,j] := \mathscr{T}_k[i,j] \cup \{Shift_k\}$, where $S_k = goto(S_i, x_j)$
   3.2. **else if** $item = [A \to \alpha\bullet]$ **then**
      3.2.1. **for** each $x_j \in \{x \mid (\omega, x\beta) \in \mathscr{C}(A \to \alpha)\}$ **do**
         if $A \in \mathscr{X}'_k$ **then**
            $\mathscr{T}_k[i,j] := \mathscr{T}_k[i,j] \cup \{Accept_{A\to\alpha}\}$
         else
            $\mathscr{T}_k[i,j] := \mathscr{T}_k[i,j] \cup \{Reduce_{A\to\alpha}\}$

Different approaches to building a PHRASE parse table can be used to reduce its size, number of conflicts, and resulting number of parsing steps. We have already employed several ideas resulting in significant reductions to the table (i.e., axiom aliased for token and consolidation of *Other* symbols). Step 3.2.1 of Algorithm 6 can reduce the number of parse actions per parse table entry at table construction time by using global information precomputed by *Las* (discussed in greater detail in Section 4.3). By considering the lookahead(1) information from all $\mathscr{C}(r)$ context sets, we avoid entering some invalid $Reduce_r$ or $Accept_r$ actions that clearly lack the proper context. It is the same method used for SLR table construction with *Follow* sets. In fact, we define $Follow(A)$ in the next section and it is equivalent to the set $\{x \mid (\omega, x\beta) \in \mathscr{C}(A \to \alpha)\}$ in Step 3.2.1 of Algorithm 6.

With all this in view the PHRASE parse table constructed by Algorithm 6 for G'_3 is shown in Fig. 7. The i in the R_i or A_i entries in Fig. 7 identifies one of the G'_3 rules, 1: $E \to T$, 2: $T \to T\ Mul\ F$, and 3: $Mul \to *$. The *Sh* in the *Mul* column of Fig. 7 is meaningless (but consistent with Algorithm 6 as written) since the only relevant information in these non-axiom columns is the *goto* state. Since axioms are aliases for

state	Terminals	Axioms (tokens)			Non-axioms	Other
	*	E	T	F	Mul	
0			Sh_1			
1	Sh_7				Sh_6	A_1 :{$, Add, +,)}
6				Sh_8		
7				R_3		R_3 :{F, (, Id, Cn, a, d, −}
8	A_2					A_2 :{$, Add, +, Mul, *,)}

Fig. 7. Parse table for G_3'.

tokens there is double meaning to a *shift* action in the axiom columns, such as Sh_8 in state 6, column F of Fig. 7. It either identifies the new state to be pushed on the stack together with the scanned input token, or it identifies the new state to be pushed on the stack together with the axiom that results from a stack reduction operation.

Algorithm 7. *PHRASE parsing algorithm for layer k of L(G)*

1. *State* := *passive*; $x :=$ *GetNextInput*( )
2. **while** $(x \neq EOF)$ **do**
   if $x \in V_k'$ **then** *Parse*$(x, Input, Stack, \mathcal{T}_k)$; *Pass*$(Stack, Output)$
   else *Pass*$(x, Output)$
   $x :=$ *GetNextInput*( )
3. Halt

Parse$(x, Input, Stack, \mathcal{T}_k, Output)$
  State := *active*
  Initialize Stack
  while (*State* = *active*) **do**
    Action := *Disambiguate*$(\mathcal{T}_k[Stack.Top.state, x])$
    if $Action = Shift_s$ then
      Stack.Push$(\langle x, s\rangle)$; $x :=$ *GetNextInput*( )
    else if $Action = Reduce_{A\to\alpha}$ **then**
      Stack.Pop$(|\alpha|)$; *Stack.Push*$(\langle A, goto(Stack.Top.state, A)\rangle)$
    else if $Action = Accept_{A\to\alpha}$ **then**
      Stack.Pop$(|\alpha|)$; *Stack.Push*$(\langle A, 0\rangle)$
      State := *passive*
    else *State* := *passive*

The PHRASE parsing algorithm is shown in Algorithm 7. The elements in the *Stack* are pairs, $\langle symbol, state\rangle$, where *symbol* and *state* have the same meaning as in LR parsing. The function *Disambiguate*( ) takes as the argument an entry in the parse table and returns the action to be performed or a failure indication. Let $\langle x_1, s_1\rangle\langle x_2, s_2\rangle \ldots \langle x_n, s_n\rangle$ be the contents of *Stack* and $Stack.Top = \langle x_n, s_n\rangle$ when $Accept_{A\to\alpha}$ is the action returned by *Disambiguate*( ). This means that $x_1x_2 \ldots x_n = x_1 \ldots$

$x_k\alpha$ for some $k \leqslant n$. An $Accept_{A\to\alpha}$ action removes states from the *Stack* to perform the reduction $x_1 \ldots x_k\alpha \overset{*}{\underset{G'}{\Longrightarrow}} x_1 \ldots x_kA$. Let v, ω be the contents of *Input* and *Output*, respectively. If we assume the initial input, γ, is in $L(G)$, then the LR parsing algorithm ensures that $\omega x_1 \ldots x_kAv$ is a reduced form of γ and furthermore that $\omega x_1 \ldots x_kA$ is a prefix of the most reduced form of γ, therefore the parser switches to passive state. The *Pass*(*Stack*,*Output*) operation causes $x_1 \ldots x_kA$ to be copied to *Output* and the *Stack* to be emptied.

Theorem 1. *If Input of the PHRASE parser is a phrase form of the language $L(G)$ then Output is also a phrase form of $L(G)$.*

Proof. The two actions performed by the parser on a given input are passing input symbols to the output and tokenizing a portion of the input whenever the viable prefix of a potential phrase specified by the grammar defining the parser is discovered. Clearly, the passing of input symbols to the output preserves correctness with respect to the parsing invariant. The tokenizing actions *Shift*, *Reduce*, and *Accept* are LR in nature and thus they preserve the parsing invariant. That is, if *Output Stack Input* is a phrase form of $L(G)$, $\langle Output, Stack, Input\rangle \overset{step}{\Longrightarrow} \langle Output', Stack', Input'\rangle$, and $Step \in \{Shift, Reduce, Accept\}$, then *Output' Stack' Input'* is a phrase form of $L(G)$. When the algorithm halts, $Stack = \varepsilon$, $Input = \varepsilon$ and thus the parsing invariant *Output Stack Input* = *Output*, i.e., *Output* is a phrase form of $L(G)$. □

4.2. SMLR parsing

A simplified PHRASE parsing method can be defined which we will call *Simple Multi-axiom LR* (*SMLR*) parsing because it is analogous to conventional SLR parsing. It differs from full multi-axiom LR parsing in that it uses only *Follow* and *Precede* sets to make parsing decisions. The *Follow* sets and the *Precede* sets can be defined in terms of the precomputed *Las* contexts as suggested in the previous section. However, we will define them in a more familiar and conventional manner here to make clear that with SMLR parsing we use a simple version of contexts rather than the full *Las* contexts.

$$Follow(N) = \{X \in \mathscr{V} \cup \{\$\} \mid A\$ \overset{*}{\Longrightarrow} \alpha NX\beta \text{ for some } A \in V\}$$

$$Precede(N) = \{X \in \mathscr{V} \cup \{\$\} \mid \$A \overset{*}{\Longrightarrow} \alpha XN\beta \text{ for some } A \in V\}$$

where \$ denotes the begin or end file markers. These definitions are similar to those used in [28]. That is, they differ from conventional definitions by including nonterminals in *Follow*(N) and *Precede*(N). We need this because axioms (as tokens) occur in the I/O streams. Fig. 8 illustrates the *Follow* and *Precede* sets for the G_3' example.

The *Follow* sets are used at parse table construction time to reduce the number of conflicting actions that Algorithm 6 sets in the parse table entries, as usual

x	$Precede(x)$	$Follow(x)$
E	$\{\$, (\}$	$\{\$, Add, +,)\}$
T	$\{\$, Add, +, (\}$	$\{\$, Add, +, Mul, *,)\}$
Mul	$\{T, F,), Id, Cn, a, d\}$	$\{F, (, Id, Cn, a, d, -\}$
F	$\{\$, Add, +, Mul, *, (\}$	$\{\$, Add, +, Mul, *,)\}$

Fig. 8. *Precede* and *Follow* sets for G_3' non-terminals.

State	Action	Output	Stack	x	Input
passive	set to *active*			$\sharp T$	$*(\sharp T + \sharp T)$
active	$Shift_1$		$\langle \epsilon, 0\rangle$	$\sharp T$	$*(\sharp T + \sharp T)$
active	$Shift_7$		$\langle \epsilon, 0\rangle\langle T, 1\rangle$	$*$	$(\sharp T + \sharp T)$
active	$Reduce_{Mul \to *}$		$\langle \epsilon, 0\rangle\langle T, 1\rangle\langle *, 7\rangle$	$($	$\sharp T + \sharp T)$
passive	*Pass Stack*		$\langle \epsilon, 0\rangle\langle T, 1\rangle\langle Mul, 6\rangle$	$($	$\sharp T + \sharp T)$
passive	*Pass x*	$\sharp T *$	$\langle \epsilon, 0\rangle$	$($	$\sharp T + \sharp T)$
active	$Shift_1$	$\sharp T * ($	$\langle \epsilon, 0\rangle$	$\sharp T$	$+\sharp T)$
active	$Accept_{E \to T}$	$\sharp T * ($	$\langle \epsilon, 0\rangle\langle T, 1\rangle$	$+$	$\sharp T)$
passive	*Pass Stack*	$\sharp T * ($	$\langle \epsilon, 0\rangle\langle E, 0\rangle$	$+$	$\sharp T)$
passive	*Pass x*	$\sharp T * (\sharp E$	$\langle \epsilon, 0\rangle$	$+$	$\sharp T)$
active	$Shift_1$	$\sharp T * (\sharp E +$	$\langle \epsilon, 0\rangle$	$\sharp T$	$)$
passive	*Pass Stack*	$\sharp T * (\sharp E +$	$\langle \epsilon, 0\rangle\langle T, 1\rangle$	$)$	ϵ
passive	*Pass x*	$\sharp T * (\sharp E + \sharp T$	$\langle \epsilon, 0\rangle$	$)$	ϵ
passive	*Halt*	$\sharp T * (\sharp E + \sharp T)$	$\langle \epsilon, 0\rangle$	EOF	ϵ

Fig. 9. $\mathscr{P}_3$ activity on $Input = \sharp T * (\sharp T + \sharp T)$.

for SLR parsing. At parse time, Algorithm 7 uses *Follow* and *Precede* sets to resolve table conflicts using function *Disambiguate*( ). The conflicting actions set by the Algorithm 6 in the parse table are resolved using the information provided by *Precede* sets. However, the SMLR algorithm performs also *Pass* actions and therefore there may be *Pass/Reduce* and *Pass/Accept* conflicts. Since *Pass* is not actually an action performed by the LR parser component of the SMLR parsing algorithm, it is not recorded in the parse table. Hence, this conflict must be implicitly associated with each $Reduce_r$ and $Accept_r$ action and is resolved by *Disambiguate*( ), using both $Follow(lhs(r))$ and $Precede(lhs(r))$. When no action has the proper context then the action is *Pass*, thus postponing the decision; another parser of the hierarchy may change the context or may find this portion of the stack to be a component of a phrase it recognizes.

Fig. 9 illustrates SMLR parsing for the layer 3 example. When T is the only stacked symbol and "+" is the next input symbol, the parser must decide whether to carry out the reduction (for the *Accept* action in this example) or to enter the *passive* state. When "(" is the preceding symbol the parser reduces T on the top of the stack to E since "$(E\ldots$" occurs in some most reduced form with respect to G_3'; but when "+" is the preceding symbol, the parser passes T since "$+E\ldots$" does not occur in any such MRF.

Since the potential phrase examined by the parser is embedded in a larger phrase, by postponing the parsing decision at conflicting points the parser preserves the correctness

of the input rather then performing changes that may affect the syntax validity of its output. Another parser of the hierarchy may perform valid reductions of the same string and thus the current phrase may be rediscovered later in a proper context. The situation is similar with that encountered by the NSLR(1) parser [28].

Theorem 2. *If a grammar is NSLR*(1) *parsable then it is SMLR parsable too.*

Proof. The parse tables for both SMLR and NSLR(1) grammars are constructed from LR(0) item sets. In addition, conflicts discovered at parse table construction time are eliminated by both NSLR(1) and SMLR parse table construction algorithms using *Follow*( ) over the entire alphabet of the language. However, while the SMLR parser can handle parse table conflicts at parsing time using *Precede*( ) over the entire alphabet of the language the NSLR(1) parser cannot operate with a parse table containing conflicts that cannot be eliminated at parse table construction. □

When correct parsing decisions can be made using *Follow* and *Precede* sets in the manner explained in this section for any layer of the hierarchy of a grammar G such that any $x \in L(G)$ can be reduced to an axiom, then we call G an *SMLR grammar* with respect to that hierarchy.

To summarize, let x be the most recent symbol of the parsing history. We assume that overloaded sentences parsed by an SMLR parser are embedded in the larger input sentences. The non-parse table conflicts $Pass/Reduce_r$ or $Pass/Accept_r$ are resolved by checking the predicate $x \in Precede(lhs(r))$. The parse table conflicts resulting from the rules r_1 and r_2 are resolved by one of the following rules:

1. If $Follow(lhs(r_1)) \cap Follow(lhs(r_2)) = \emptyset$ then there will be no table conflicts associated with r_1 and r_2 because any potential conflict is statically resolved by Algorithm 6.
2. If $Follow(lhs(r_1)) \cap Follow(lhs(r_2)) \neq \emptyset$ but $Precede(lhs(r_1)) \cap Precede(lhs(r_2)) = \emptyset$ then there will be table conflicts and they will be resolved dynamically by Algorithm 7 checking the predicates $x \in Precede(lhs(r_1))$ and $x \in Precede(lhs(r_2))$.
3. If $Follow(lhs(r_1)) \cap Follow(lhs(r_2)) \neq \emptyset$ and $Precede(lhs(r_1)) \cap Precede(lhs(r_2)) \neq \emptyset$ then if $x \notin Precede(lhs(r_1)) \cap Precede(lhs(r_2))$ the table conflict is resolved similar to rule (2) above, otherwise the conflict cannot be resolved and the action is *Pass*.

When conflicts cannot be resolved by the use of simple *Precede* and *Follow* sets a more precise collection of preceding and following contexts is required. Thus, the relation between SMLR and this more powerful method, described in the next section, is analogous to that of conventional SLR and LR.

4.3. Conflict resolution by LookAround method

This section describes the *LookAround* method that resolves parse table conflicts dynamically, during PHRASE parsing. For that we assume that the preprocessing of the initial grammar is done to collect context information about potential conflict points. When simple *Follow* and *Precede* sets are inadequate, then a more powerful form of

this information is computed which consists of contexts [22, 23] associated with the rules, rather than the non-terminals. For each rule, one computes the set of contexts which may surround the collection of phrases specified by the rule. That is, for a given rule r of the form $M \rightarrow \omega$, $\mathscr{C}(r)$, $\mathscr{N}(r)$, and $\mathscr{A}(r)$ are sets of pairs of strings (x, y) of variable-length m, n, respectively, referred to as (m, n) contexts, that have the following properties:

1. if $(x, y) \in \mathscr{C}(r)$ then r has been used in a derivation to generate the phrase $\alpha\ x\ \omega\ y\ \beta$ from a syntactically valid phrase of the form $\alpha\ x\ M\ y\ \beta$.
2. if $(x, y) \in \mathscr{N}(r)$ then r has not been used in a derivation to generate the phrase $\alpha\ x\ \omega\ y\ \beta$ from a syntactically valid phrase of the form $\alpha\ x\ M\ y\ \beta$.
3. if $(x, y) \in \mathscr{A}(r)$ then one cannot decide if r was used in the derivation of the phrase $\alpha\ x\ \omega\ y\ \beta$, respectively, from a syntactically valid phrase of the form $\alpha\ x\ M\ y\ \beta$.

That is, from the parsing viewpoint the pair $(x, y) \in \mathscr{C}(r)$ has the property that whenever a syntactically valid input string has the form $\alpha x \omega y \beta$ it can be reduced to $\alpha x M y \beta$ preserving its syntactic validity; if $(x, y) \in \mathscr{N}(r)$ then the input $\alpha x \omega y \beta$ cannot be reduced to $\alpha x M y \beta$; if $(x, y) \in \mathscr{A}(r)$ the decision should be postponed. Further, we refer to the tuple $\langle \mathscr{C}(r), \mathscr{N}(r), \mathscr{A}(r) \rangle$ by *Context*(r).

If a conflict is associated with parsing actions involving the rules $r_1 : A \rightarrow \alpha$ and $r_2 : B \rightarrow \alpha$ then rules r_1 and r_2 will have different contexts associated with them. During parse table construction, when a *reduce/reduce*, *reduce/accept*, or *accept/accept* conflict is detected, the contexts identifying $[A]$ and $[B]$ are attached to the table entry containing the conflict. During a parse the parser makes use of this additional precomputed information by means of the *LookAround* method [25], a formal means of conflict resolution that consists of two actions, *LookAhead* and *LookBack*. Let *rhs*(r) be on the top of the stack for some r when a conflict point is reached. To determine the appropriate action the parser uses *LookAhead* and *LookBack* to examine the current context of *rhs*(r) and compare it with the precomputed *Context*(r), for all rules involved in the conflicting actions. For each (m, n)-context in *Context*(r) at most n lookahead and m lookback symbols must be examined. The *LookAhead* action examines the next symbols in the input. It is the same as with any LR(k) parser, is well understood and needs no further comment. The *LookBack* action examines the previous symbols in the *stack history* (that is, the stack together with those symbols already output) and its use is explained as follows: When a *reduce/reduce*, *reduce/accept*, or *accept/accept* conflict for productions $A \rightarrow \alpha$ and $B \rightarrow \alpha$ is identified, the *Context*$(A \rightarrow \alpha)$ and *Context*$(B \rightarrow \alpha)$ are examined. If the most recent *stack history* is γ and (γ, δ) is in *Context*$(A \rightarrow \alpha)$ or *Context*$(B \rightarrow \alpha)$ for some δ, then

1. $(\gamma, \delta) \in \mathscr{C}(A \rightarrow \alpha)$ implies that α can be reduced to A preserving syntax validity.
2. $(\gamma, \delta) \in \mathscr{N}(A \rightarrow \alpha)$ implies that α cannot be reduced to A preserving syntax validity.
3. $(\gamma, \delta) \in \mathscr{C}(B \rightarrow \alpha)$ implies that α can be reduced to B preserving syntax validity.
4. $(\gamma, \delta) \in \mathscr{N}(B \rightarrow \alpha)$ implies that α cannot be reduced to B preserving syntax validity.

Since $\mathscr{C}(A \rightarrow \alpha) \cap \mathscr{C}(B \rightarrow \alpha) = \emptyset$ and since for unambiguous grammars, for each production r, $\mathscr{C}(r) \cap \mathscr{N}(r) = \emptyset$, these rules resolve the conflict. The information γ required by the *LookBack* action is collected at parse table construction time and only as much of

the context as is needed to resolve a conflict is recorded. That is, in the above explanation, the sets *Precede*(A) and *Precede*(B) are initially computed and their intersection is taken. If the intersection is empty, the conflict is resolvable by a *LookBack*(1) action. If the intersection is not empty, *LookBack*(2) is computed by extending the symbols in the intersection to two symbol strings by concatenating them to the left with their *Precede*. The process continues until the conflict is resolvable for all contexts. In this way the k involved in any *LookBack* action is always the smallest possible.

Shift/reduce and *shift/accept* parse table conflicts fall into two categories. If the conflict is based on the two items $[A \rightarrow \alpha\bullet]$ and $[B \rightarrow \alpha \bullet \beta]$, then it is resolvable by either a *LookBack* or *LookAhead* action as previously described. If the conflict is based on the two items $[A \rightarrow \alpha\bullet]$ and $[A \rightarrow \alpha \bullet \beta]$, then the conflict is resolvable by a *LookAhead* action.

In essence the *LookBack* action is a pattern-matching algorithm that may use the entire stack history as a string to be matched by the patterns precomputed at parse-table construction time. Because this includes a portion of the input which has already been tokenized, the *LookBack* action performs a context check using a finite context that may represent a potentially infinite number of strings of potentially infinite length in the text. Therefore, it is extremely powerful.

To summarize, the *LookAround* method makes use of the complete state of the parsing machine. The entire stack history for a given pass, if necessary, can be examined in order to resolve a parsing conflict, or additional input symbols can be inspected, if needed. Note that this is not the same as an LR(k) parser for an arbitrary k. In such a parser the entire k-lookahead table must be computed and used by the parsing machine. In the *LookAround* method the parse table is computed for an SLR parser and conflicts are resolved by additional context necessary for performing the actions *LookBack* (which examines the stack history) and *LookAhead* (which examines the input). This information is precomputed and recorded during the parse table building process only at points of conflict. The number of *LookBack* or *LookAhead* contexts are determined by each separate conflict, and only the contexts necessary to resolve a conflict are computed and attached to the parse table.

The PHRASE parser with *LookAround*, then, is a variable k parser rather than a fixed k parser. At each point in the parse table the smallest k necessary to provide a deterministic parse is computed, and only that necessary information is retained. Usually this k is 0. Only a small number of entries need a k equal to or greater than 1, and then the k is still a small number. Such a parser can be efficiently implemented.

Theorem 3. *The class of grammars parsable by PHRASE parser with LookAround conflict resolution includes all non-ambiguous context-free grammars.*

Proof. As shown in [14], if a grammar, G, is not ambiguous then $\mathscr{A}(r) = \emptyset$ for each $r \in P$. That is, $\mathscr{C}(r) \cap \mathscr{N}(r) = \emptyset$ for all $r \in P$. In other words the *LookAround* method can always find a unique action to be performed irrespective of the number of conflicts recorded in the parse table entries. □

5. Multi-pass parsing

In this section we present the parsing algorithm which formally captures the tokenization process associated with language layers and expands it into the multi-layered multi-pass parsing algorithm. The goal of this algorithm is to recognize the primary phrase which the entire input stream represents. Parsing by layer occurs in left-to-right fashion, but the overall process is multi-pass parsing which can naturally be pipeline parallelized for speed. This process is illustrated in Fig. 10.

Each parser function can be thought of as a filter which transforms the input piped through it. The diagram shows loops back to previous stages. In the conventional LR parser, such loops are contained within the parser as one observes in the FSM of item sets which usually has many cycles. We have fragmented the parse tables and therefore may have cycles in the pipeline as well as within the individual parsers. However, not every parsing stage needs to have a loop back to a previous stage, nor does such a loop need to return to the start. The nature of the pipeline loops depends on the characteristics of the stratified grammar and the characteristics of a particular input. Algorithm 4 can be changed to compute the *repetition coefficients* [14, 22] that can be used to minimize these loops. The algorithm presented here is a general one that makes no attempt to determine an optimal pipeline path for the input but rather takes a worst case, guaranteed approach.

Let G be the multi-axiom grammar specifying a programming language and G' a primitive subgrammar of G. The multi-pass pipeline parser is constructed as follows:

1. From the given G and G', compute $(G'_0, G_0), (G'_1, G_1), \ldots, (G'_m, G_m)$, by Algorithm 4.
2. For $i = 0, 1, \ldots, m$, construct the parse table $\mathscr{T}_i$ from the primitive subgrammar G'_i by Algorithm 6.
3. Create instances of Algorithm 7 for each layer, $i = 0, 1, \ldots, m$, and let $\mathscr{P}_i$ be the PHRASE parser for layer i which references table $\mathscr{T}_i$.
4. Input strings are processed by the sequence $\mathscr{P}_0, \mathscr{P}_1, \ldots, \mathscr{P}_m$ of PHRASE parsers used repeatedly in a pipeline fashion, until no transformation of input occurs. This is shown by Algorithm 8 in which the UNIX pipe operator, "|", denotes the piping from layer to layer.

Algorithm 8. *Multi-pass pipeline parser*

repeat *forever*

pipe input *through* $\mathscr{P}_0|\mathscr{P}_1|\ldots|\mathscr{P}_m$ *transforming it to* output

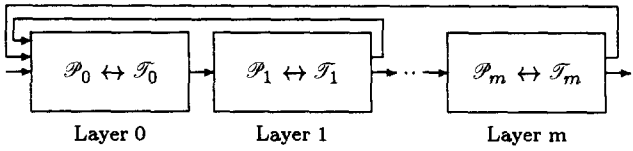

Fig. 10. Multi-layered pipeline parsing.

Parser	Parser Output
initial input	$d + (a + -dd) * aa * dd$
$\mathscr{P}_0$	$\sharp Cn + (\sharp Id + \sharp Cn) * \sharp Id * \sharp Cn$
$\mathscr{P}_1$	$\sharp F + (\sharp F + \sharp F) * \sharp F * \sharp F$
$\mathscr{P}_2$	$\sharp T + (\sharp T + \sharp T) * \sharp F * \sharp F$
$\mathscr{P}_3$	$\sharp E + (\sharp E + \sharp T) * \sharp F * \sharp F$
$\mathscr{P}_4$	$\sharp E + \sharp F * \sharp F * \sharp F$
$\mathscr{P}_2$	$\sharp E + \sharp T * \sharp F * \sharp F$
$\mathscr{P}_3$	$\sharp E + \sharp T$
$\mathscr{P}_4$	$\sharp E$

Fig. 11. Multi-pass parsing of an expression.

if input = output **then** *exit loop*
else *let* input = output

Each $\mathscr{P}_i$ in this algorithm is a PHRASE parser. The complexity of the Pass actions performed by the PHRASE parser is obviously $\mathcal{O}(n)$ where n is the length of the input. In addition, according to Theorem 5.13 in [2] the complexity of the SLR component of the PHRASE parser is also $\mathcal{O}(n)$. Hence, the complexity of $\mathscr{P}_i$ is $\mathcal{O}(n)$. In our case each input string of $\mathscr{P}_i$ is either an element of the primitive language generated by G_i' or a primitive symbol that gets passed. The number of phrases recognized by all $\mathscr{P}_i$ is at most the number of axiom nodes within the derivation tree of the entire input string. If the input string has the length n then this number is $\mathcal{O}(\log n)$. That is, the average length of a phrase recognized by $\mathscr{P}_i$ is $\mathcal{O}(n/\log n)$. This means that the complexity of each action performed by $\mathscr{P}_i$ is $\mathcal{O}(n/\log n)$. Since there are $\mathcal{O}(\log n)$ such actions to recognize the entire string, the complexity of the entire parse is $\mathcal{O}(\log n) * \mathcal{O}(n/\log n) = \mathcal{O}(n)$. But when the pipe is full, each parser $\mathscr{P}_i$, $0 \leqslant i \leqslant m$, processes one of the nodes of the derivation tree in parallel, that is the complexity of the piping-algorithm is $\mathcal{O}(n/(m+1))$. Note, we disregard here the time required by conflict resolution and pipe synchronization.

Consider again the expression grammar which was stratified (Fig. 3). The table in Fig. 11 summarizes the results of an optimal pipeline path for the given simple input.

Algorithm 8 is a brute force one since there can be many useless passes over the input by individual PHRASE parsers (i.e., those that leave the input unchanged). The algorithm terminates when every one of the $m+1$ passes in an iteration is useless, which is an inefficient, but correct, way of knowing that the input has been tokenized as much as it can be. Ideally we want to predict which PHRASE parsers will be useful and only include those in the pipeline operation. For instance, if we have a 4-layer hierarchy in which all axioms in G_0' are completely defined in the subgrammar and likewise for G_1', then the parse sequence $\mathscr{P}_0,\mathscr{P}_1,\mathscr{P}_2,\mathscr{P}_3$, $\mathscr{P}_0,\mathscr{P}_1,\mathscr{P}_2,\mathscr{P}_3,\ldots,\mathscr{P}_0,\mathscr{P}_1,\mathscr{P}_2,\mathscr{P}_3$ can be reduced to $\mathscr{P}_0,\mathscr{P}_1,\mathscr{P}_2,\mathscr{P}_3,\mathscr{P}_2,\mathscr{P}_3,\ldots,\mathscr{P}_2,\mathscr{P}_3$. Some research has been done in this

Rule	Context	Noncontext	Ambiguity
$\bar{A} \to c$	$\{(\$, \bar{A}), (\$, A), (c, A), (c, a), (c, \bar{A})\}$	$\{(\$, B), (c, B), (c, b), (\bar{B}, B)\}$	$\{(c^n, \$)\}$
$\bar{B} \to c$	$\{(\$, \bar{B}), (\$, B), (c, B), (c, b), (c, \bar{B})\}$	$\{(\$, A), (c, A), (c, a), (\bar{A}, A)\}$	$\{(c^n, \$)\}$
$A \to \bar{A}$	$\{(\$, a), (\bar{A}, a), (c, \$), (c, a)\}$	$\{(\$, A), (c, A), (\$, c), (c, c)\}$	$\emptyset$
$B \to \bar{B}$	$\{(\$, b), (\bar{B}, b), (c, \$), (c, b)\}$	$\{(\$, B), (c, B), (\$, c), (c, c)\}$	$\emptyset$
$A \to \bar{A}A$	$\{(\$, a), (\bar{A}, \$), (\bar{A}, a), (c, \$), (c, a)\}$	$\emptyset$	$\emptyset$
$B \to \bar{B}B$	$\{(\$, b), (\bar{B}, \$), (\bar{B}, b), (c, \$), (c, b)\}$	$\emptyset$	$\emptyset$
$S \to Bb$	$\{(\$, \$)\}$	$\{(c, \$), (\bar{B}, \$)\}$	$\emptyset$
$S \to Aa$	$\{(\$, \$)\}$	$\{(\bar{A}, \$), (c, \$)\}$	$\emptyset$

Fig. 12. Complete context information for a non-LR(k) grammar.

area with algebraic grammars [14] but more study is required for general case multi-axiom grammars.

Theorem 4. *The class of grammars that are parsable by multi-pass PHRASE parsers strictly includes the LR(k) grammars.*

Proof. The inclusion of LR(k) grammars in the class of grammars parsable by PHRASE parsers is a consequence of Theorem 3. To show that the inclusion is strict we construct a PHRASE parser for the language specified by the CFG grammar $G = \langle \{\bar{A}, \bar{B}, A, B, S\}, \{a, b, c\}, \{\bar{B} \to c, \bar{A} \to c, A \to \bar{A}, A \to \bar{A}A, B \to \bar{B}, B \to \bar{B}B, S \to Bb, S \to Aa\}, \{S\} \rangle$ which is not LR(k) for any k [28]. Applying *Las* on the grammar G we get the computed context shown in Fig. 12.

In this case, the smallest primitive subgrammar of G is G itself which means there is only one language layer. Thus, there is only one PHRASE parser for $L(G)$. The problematic situation that exists for the LR(k) parser is that one cannot decide whether to reduce c to $\bar{A}$ or $\bar{B}$ without looking ahead to see if the last input symbol is a or b, respectively. The multi-axiom parser has no problem with this since it is not limited to a fixed k of lookahead, as is the LR(k) parser. For the PHRASE parser it is the non-empty ambiguity sets which can be problematic.

The only non-empty ambiguity sets here (see Fig. 12) is when the input is a stream of c's terminated by an EOF, not by an a or b. Since this is not a valid phrase in $L(G)$ the multi-axiom parser will pass the input unchanged to the output stream, no matter how the table conflict is resolved initially (thus, it is resolved as a pass action). Since the *LookAround* method resolves any other table conflicts, G is a grammar parsable by PHRASE parsers. □

6. Further work and final comments

SMLR PHRASE parsing has been implemented and demonstrated to show the independent operation of individual parsing layers [24]. SMLR parsing was added as a feature of a menu driven system designed for studying multi-axiom grammars, called MAGLAB (Multi-Axiom Grammar LAB environment). Since *error* actions are han-

dled as *pass* actions, error handling needs to be implemented. One possibility is to develop a mechanism which monitors the overall error-checking process separate from and parallel with the PHRASE parsers. However, from the error discovery and recovery viewpoint the manner of postponing parsing decisions by PHRASE parsers is translated in a strategy where parsing continues by automatic recovery from erroneous states, recording their positions in the input and by propagating these positions to the end of the parsing. Then when parsing completes without recognizing the input as valid the parser can display complete and correct information about its potential erroneous parts.

Our current implementation does use a micro-scanner as an initial pass to get word strings which are atomic according to how the BNF rules are written. In practice, some form of micro-scanner is needed (below our layer 0) to act as a character classifier at least. We have also ignored making a distinction between rules whose right-hand sides allow whitespace and those that do not (i.e., lexeme rules), but it is clear that such distinction is needed. One possibility is to introduce special whitespace or glue (i.e., non-whitespace) terminals.

Further expanding the role of non-axioms to allow them to reach axioms may be useful to consider, providing that the notion of primitivity and language layering based on the smallest primitive subgrammar are reconsidered. Another idea worth consideration is to allow the language designer to choose a primitive subgrammar other than the smallest one for each grammar layer.

In summary, we have shown how MAGs can be the specification tool of choice for language design. Using MAGs the language designer can approach the task of language design as both algebraist and systems engineer. The algebraist identifies which variables of the overall specification are axioms, representative of the true phrases of the language and considers how the choice of axioms and axiom rules affect the language hierarchy. The systems engineer uses non-axioms to represent intermediate constructs in order to achieve readability and modularity in the BNF specification.

We have shown how to construct parsing mechanisms in accordance with the algebraic properties and structure of the language. Such parsers can accept stand-alone phrases as well as complex sentences and do so in parsing stages, in correspondence with language layers, which are independent so that a loosely coupled pipeline can be implemented.

Acknowledgements

We thank the anonymous reviewers for their careful reading of this paper and for valuable suggestions that helped us make many improvements of the original version. These improvements extend from making the paper more readable, to comparing our research with other similar ideas used in parsing technology, such as [19, 26, 28], and finally to providing a first evaluation of the class of languages parsable by our methodology.

References

[1] A.V. Aho, R. Sethi, J.D. Ullman, Compilers: Principles, Techniques, and Tools, Addison-Wesley, Reading, MA, 1986.

[2] A.V. Aho, J.D. Ullman, The Theory of Parsing, Translation, and Compiling, vol. I: Parsing, Prentice-Hall, Englewood Cliffs, NJ, 1972.

[3] A.V. Aho, J.D. Ullman, Principles of Compiler Design, Addison-Wesley, Reading, MA, 1977.

[4] J. Bates, A. Lavie, Recognizing substrings of LR(k) languages in linear time, ACM Trans. Programming Languages Systems 16 (3) (1994) 1051–1077.

[5] L. Boasson, Dérivations et réductions dans les gramaires algébriques, in: Proc. 7th ICALP, Lecture Notes in Computer Science, vol. 85, Springer, Berlin, 1980, pp. 109–118.

[6] F.L. DeRemer, Practical Translators for LR(k) Languages, Ph.D. Thesis, MIT, Cambridge, MA, 1969.

[7] F.L. DeRemer, Simple LR(k) grammars, Comm. ACM 14 (7) (1971) 453–460.

[8] J. Earley, An efficient context-free parsing algorithm, Comm. ACM 4 (2) (1970) 183–192.

[9] R.W. Floyd, Syntactic analysis and operator precedence, J. ACM 10 (3) (1963) 316–333.

[10] R.W. Floyd, Bounded context syntactic analysis, Comm. ACM 7 (2) (1964) 62–67.

[11] W.S. Hatcher, T. Rus, Context-free algebras, Cybernetics 6 (2–3) (1976) 65–76.

[12] J.E. Hopcroft, J.D. Ullman, Introduction to Automata Theory, Languages, and Computations, Addison-Wesley, Reading, MA, 1979.

[13] J.S. Jones, Multi-Layered Pipeline Parsing of Phrases from Multi-Axiom Grammars, Ph.D. Thesis, University of Iowa, Department of Computer Science, Iowa City, IA 52242, 1997.

[14] J. Knaack, The Algebraic Approach to Compilation, Ph.D. Thesis, University of Iowa, Department of Computer Science, Iowa City, IA 52242, 1994.

[15] D.E. Knuth, On the translation of languages from left to right, Inform. and Control 8 (6) (1965) 607–639.

[16] P.M. II Lewis, R.E. Stearns, Syntax-directed transductions, J. ACM 15 (3) (1968) 465-488.

[17] W.M. McKeeman, Compiler construction, in: F. Bauer, J. Eickel (Eds.), Compiler Construction, Springer, Berlin, 1976, pp. 1–36.

[18] A. Nijholt, Computers and Languages, Studies in Computer Science and Artificial Intelligence, vol. 4, North-Holland, Amsterdam, 1988.

[19] A. Nijholt, The CYK-approach to serial and parallel parsing, Language Res. 27 (2) (1991) 229–254.

[20] T. Rus, σs-algebra of a formal language, Bull. Math. de la Soc. de Sci., Bucharest, 15 (63) (2) (1972) 227–235.

[21] T. Rus, Context-free algebra: a mathematical device for compiler specification, in: Lecture Notes in Computer Science, vol. 45, Springer, Berlin, 1976, pp. 488–494.

[22] T. Rus, Parsing languages by pattern matching, IEEE Trans. Software Eng. 14 (4) (1988) 498–510.

[23] T. Rus, T. Halverson, Algebraic tools for language processing, Comput. Languages 20 (4) (1994) 213–238.

[24] T. Rus, J.S. Jones, Multi-layered pipeline parsing from multi-axiom grammars, in: A. Nijholt, G. Scollo, R. Steetskamp (Eds.), Algebraic Methods in Language Processing, AMiLP'95, University of Twente, Department of Computer Science, NL 7500, AE Enschede, 1995, pp. 65–81.

[25] T. Rus, J. LePeau, Language specification by multi-axiom grammars, in: Proc. Internat. Conf. on Computer Languages, IEEE Computer Society Press, Silver Spring, MD, 1988, pp. 110–118.

[26] D.J. Salomon, G.V. Cormack, Scannerless NSLR(1) parsing of programming languages, SIGPLAN Notices 24 (7) (1989) 170–178.

[27] F. Southworth, C. Daswani, Foundations of Linguistics, The Free Press, New York, 1976.

[28] K.C. Tai, Noncanonical SLR(1) grammars, ACM Trans. Programming Languages and Systems 1 (2) (1979) 295–320.